平成 29 年度修士論文

監視対象システムを止めずにカーネル制御フロー改変ルートキットを検知するシステム

電気通信大学
大学院情報理工学研究科
情報・ネットワーク工学専攻
コンピュータサイエンスコース

学籍番号　：　1631068
氏名　：　徐 振宇
主任指導教員　：　岩崎 英哉 教授
指導教員　：　沼尾 雅之 教授
提出日　：　2018年1月29日

要旨

カーネル制御フロー改変ルートキットとは，ユーザプロセスが発行するシステムコールのカーネル内処理ルーチンを改竄し，目的を実現するマルウェアである．この種のルートキットは，実現しやすい，汎用性が高い，そして検知されにくいという特徴を持つ．ルートキットにより汚染されたシステムの上で動作するルートキット検知システムの挙動は信用できないため，検知システムは仮想マシンモニタを利用するなどして，対象システムの外部に置くのが一般的である．

しかし，対象システムを外部に置くと，セマンティクギャップという問題が生じる．従来の多くの研究では，一時的に対象システムを止めて，システムから必要な情報を取得する手法を用いてセマンティクギャップ問題を解決する．しかし，この手法では，オーバーヘッドが大きく，対象システムのパフォーマンスを低下させる問題点が生じる．

本研究では，対象システムを止めることなく，監視対象の外部からルートキットを検知するシステムを目指す．本システムは対象システムのカーネル関数の呼び出し履歴 (トレース情報) を取得するカーネル組み込みの Ftrace を拡張して利用する．更に，Ftrace は対象システム内部で動作しているため，トレース結果がルートキットにより改竄される恐れがある．本研究では，ルートキットによるトレース情報の改竄を困難にするため，トレース情報をホスト側で用意した鍵と XOR を取るように対象システムを拡張した．これらの目的を達成するため，本研究では，Ftrace と QEMU-KVM 及びゲスト OS のカーネルを拡張し利用した．

目次

1 はじめに

1.1 研究の背景

ルートキットとは，攻撃者が対象システムに対し，目的に応じたプログラムを埋め込むためのマルウエアである．ルートキットの特徴は，対象システムに潜伏し，攻撃者が自らの攻撃をしやすくするようにバックドアを構築し，自らの存在を隠すことである．

カーネル制御フロー改変ルートキット (kernel control-flow attacks rootkit) とは，ユーザプロセスが発行するシステムコールのカーネル内処理ルーチンを改竄し，目的を実現するルートキットである．カーネル制御フロー改変ルートキットは，実現しやすい，汎用性が高い，そして検知されにくいという特徴を持つため，もっともよく用いられる攻撃の手法である [4]．本研究はカーネル制御フロー改変ルートキット (以下単にルートキットと呼ぶ) を検知の対象として，検知システムを設計する．本システムは xftrace と名付ける．

図 1.1 に `read` システムコールをターゲットとした攻撃の例を示し，これを用いてルートキットの動作について説明する．この例では，攻撃者はカーネル空間内のシステムコールテーブルにある `read` システムコールのカーネル内処理ルーチンである `sys_read` 関数の場所のアドレスを書き換えることによって，攻撃を仕掛けている．ユーザプロセスが `read` システムコールを発行すると，割り込み命令によりカーネル空間のシステムコールディスパッチャに制御が移る．システムコールディスパッチャは，発行されたシステムコールの番号を添字としてシステムコールテーブルを参照する．システムコールテーブルには各システムコールに対するカーネル内処理ルーチンの先頭アドレスが入っており，`read` の場合はカーネル内関数 `sys_read` の先頭アドレスになっている．図 1.1 では，攻撃者がこれを自身で用意した攻撃用の関数 `evil_read` の先頭アドレスに書き換えている．そのため，攻撃者が用意したこの関数が実行されてしまう．攻撃者は，`evil_open` の中に重要な情報を盗む等の攻撃コードを含ませることによって，目的を達成する．

攻撃者に狙われやすいシステムコールは，普段よく使われるシステムコールである [3]．例えば，ファイル操作のシステムコール (`open`，`read`，`write`，`close`，`mkdir`，`getdents`)，プロセス操作のシステムコール (`fork`，`exec`，`clone`，`kill`，`setuid`)，ネットワーク操作のシステムコール (`socket`，`connect`，`bind`，`listen`，`accept`) がよく狙われる．

また，システムコールテーブル以外にも狙われやすい標的がある．たとえば，SucKIT [10] ルートキットは Interrupt Descriptor Table を標的とし，Adore-ng [5] ルートキットは File Operations Table を標的とする．

ルートキットを検知するシステムのアプローチには，一般的に以下の 3 つがある．

1 つ目は，マルウェアのシグネチャを利用するアプローチ [1, 2] である．このアプローチは，

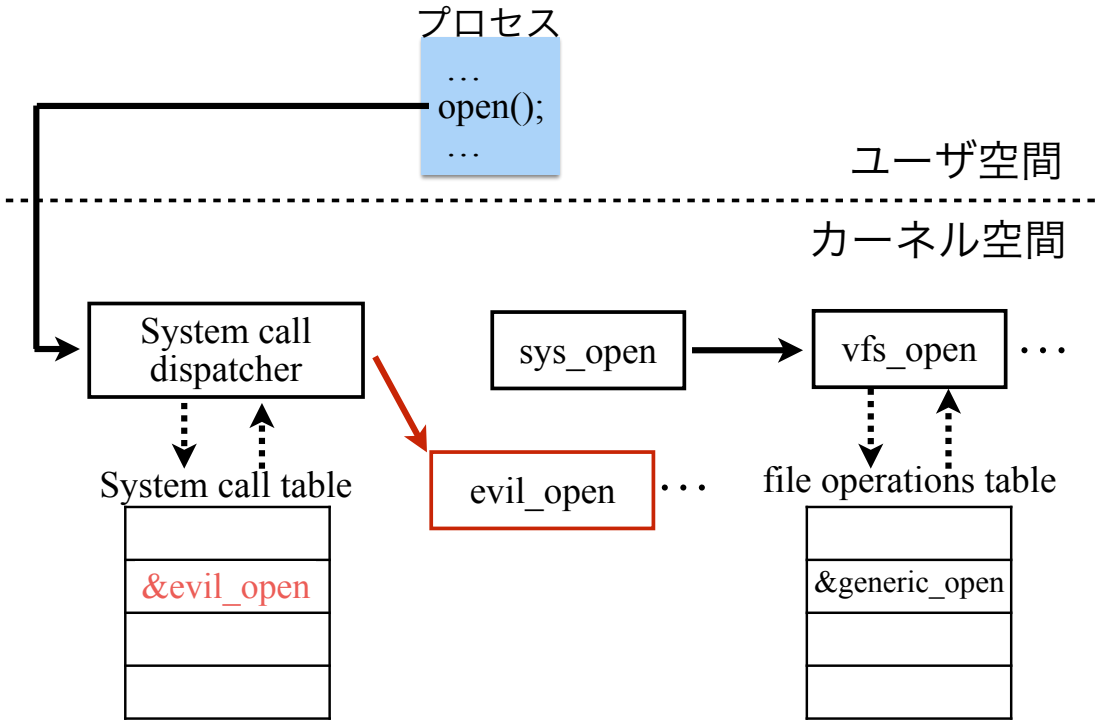


図 1.1　カーネル制御フロー改変ルートキットの例

対象システム上にあるファイルを全部スキャンして，マルウェアのコードに特有のバイト列が含まれているどうかを判別する．多くのアンチウィルスソフトウェアは，このアプローチを採用している．しかし，対象システムがルートキットに既に汚染され，検知システムが対象システム上で動作する場合は，検知システムが無効化される恐れがある．また，サイバーセキュリティで有名なファイルレス攻撃 [6] の検知は，このアプローチでは検知できない．その理由は，ファイルレス攻撃では，マルウェアの実体が対象システムのメモリ上にあり，ファイルとして存在しないからである．

2 つ目は，差異に基づく検知アプローチ [17, 20] である．このアプローチは，対象システム上にあるバイナリを，事前に入手した信頼できるバイナリと比較することにより，ルートキットを検知する．対象システムごとに検知に必要な情報を，検知システム側に持つ必要がある．例えば，隠されたプロセスを検出するため，検知システム側で対象システム上にあるプロセスリストを全部外で再現する必要がある．

3 つ目は，挙動ベースの検知アプローチ [18] である．このアプローチは，システムの振る舞いを観察し，ルートキットに汚染されていない時と異なる振る舞いを検知することによって，ルートキットの存在を推測する方法である．この方法には，上記の方法のような弱点がなく，また対象システムに与える影響が少ないため，本研究はこのアプローチを採用する．さらに本研究では，汚染されている可能性のある対象システムから検知システムを隔離するため，仮想マシンを用いて，対象システムの外部から検知する．

仮想マシンを用いると，検知システムは対象システムの外にいるため，対象システムを一時的に止めて，検知システムが必要な情報を取得する方法が一般的に用いられる．しかし，対象システムを止める必要があるため，対象システムのパフォーマンスを低下させてしまう．

1.2 目的と方針

本研究の目的は，対象システムを止めず，外部からカーネル制御フロー改変ルートキットを検知するシステム xftrace の実現を目指すことである．

本研究の方針は以下の 3 つとする．1 つ目は，Linux KVM [9] 仮想マシンを用いることである．対象システムをゲスト OS とし，検知システムをホスト側で実現する．2 つ目は，ユーザプロセスによるシステムコールの発行時に，カーネル内の関数呼び出し履歴を収集することである．その理由は，3.3 章で述べるように，カーネル内の関数呼び出し履歴を分析することは，ルートキットの発見に有効であることが知られている [18, 19, 20] からである．3 つ目は，ルートキットによるトレース結果の改竄を困難にすることである．Ftrace がゲスト OS 内で動作するため，ルートキットによるトレース情報の改竄を難しくするため，トレース情報をホスト側で用意した鍵と排他的論理和 (XOR) を取るようにゲスト OS を拡張し，関数呼び出し履歴をホスト側で生成するようにシステムを設計し実装した．また，本システムの設計は，改竄を難しいくすることを目指しており，改竄を不可能するものではない．本研究では，この目的を達成するため，Ftrace [7] と QEMU-KVM [8] 及びゲスト OS のカーネルを拡張し利用する．

1.3 本論文の構成

本論文の構成は次のとおりである．まず，第 2 章で既存システムの問題点を述べる．第 3 章で，Ftrace の説明と Ftrace を用いる理由を述べる．第 4 章では，Ftrace の問題点やシステムの設計を説明し，ユーザの利用方法を述べる．第 5 章で本システム実装方法を説明し，第 6 章で本システムの評価と議論を行う．第 7 章で関連研究を述べる．第 8 章で本研究のまとめと今後の課題について述べる．

2　既存システムの問題点

本章では，既存システムの例を挙げて，その概要と問題点を説明する．

2.1　従来の方式

仮想マシンを用いて，検知システムを対象システムの外に設置すると，セマンティックギャップ [11] 問題の発生により，検知システムの性能が制限される．セマンティックギャップ問題とは，検知システムが直接アクセスできる CPU レジスタやメモリ内容などの低いハードウェアレベルの情報と，ゲスト OS のプロセスやカーネル内で発生したイベンドなどの高いレベルの情報との間にはギャップがあるという問題である．したがって，得られる低レベルの情報から高レベルの情報を推測しなければならない．検知システムが対象システムから情報を得るために一般的に用いられる手法は，対象システムを一時的に止めることである．

VSyscall の例

VSyscall [12] を例として説明する．ゲスト OS 内のプロセスにより呼び出されたシステムコールは，仮想マシンモニタ (VMM) に検知される．VMM は，ゲスト OS の実行状態を直接見ることができないため，ゲスト OS のシステムコールの実行を識別する．その後，VSyscall はゲスト OS から呼び出されたシステムコールイベントを解釈し，システムコールがどのプロセス

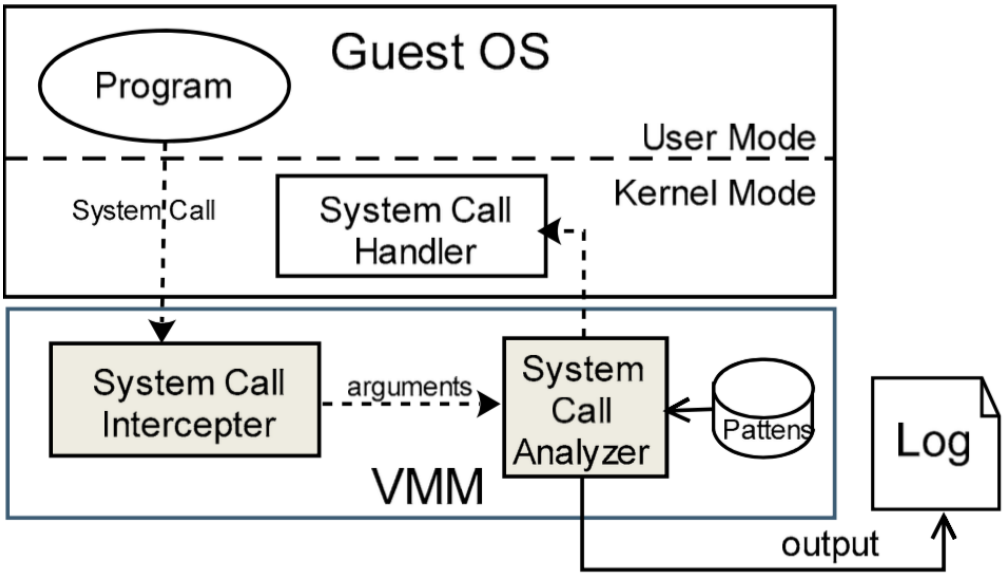


図 2.1　VSyscall システムの設計

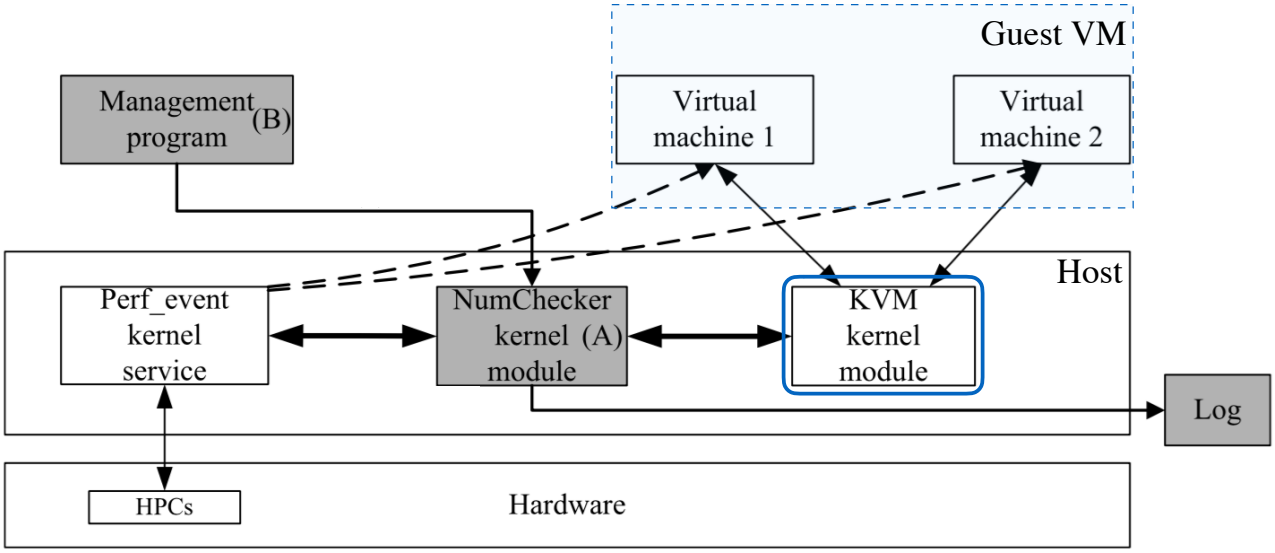


図 2.2　NumChecker システムの設計

に属しているかを調べ，システムコールとの関係を確立する．これにより，プロセスの動作を監視する．

図 2.1 に示す通り，VSyscall には，System Call Interpreter (SCI) と System Call Analyzer (SCA) の 2 つ重要なコンポーネントがある．SCI は，ゲスト OS のユーザモードにおいて実行されたシステムコール命令 (int 80h または sysenter) を検知し，システムコールの引数を取得して SCA に渡す．SCA はこれらの情報からシステムコール間の関係を確立し，与えられた正常時のシステムコール呼び出しのパターンに基づいてプログラムの動作を監視する．システムコールの呼び出し順とのパターンマッチング結果はログに出力し，ルートキットの分析に用いる．

このシステムの欠点は SCI にある．SCI は，制御をゲスト OS から VMM に移すため，システムコールが呼び出される時，カーネル内処理ルーチン関数のアドレスを存在しないアドレスに置き換え，ページフォールトを起こす．ページフォールトに対する処理には，大きなオーバーヘッドがかかる．このようにして，対象システムを止める必要があるため，対象システムの性能が損なわれてしまう．

Numchecker の例

NumChecker [14] は，ハードウェアパフォーマンスカウンタ (HPCs) を利用し，システムコール実行時に実際に行った機械語命令をカウントする．

NumChecker には，図 2.2 に示す 2 つの重要な部品がある．部品 A では，KVM から監視したシステムコールの情報をとり，更に Pref をコントロール[*1]する役目を果たしている．部品 B

[*1] HPCs 初期化や記録タイミングなど．

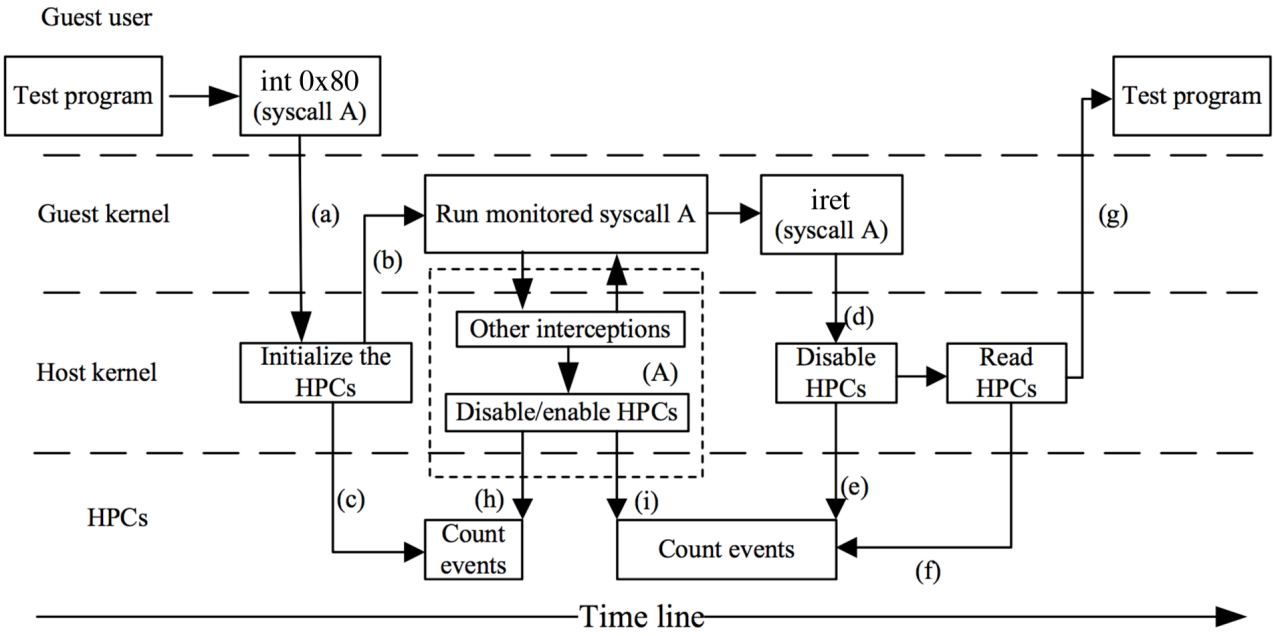


図 2.3　NumChecker システムの動作

は，NumChecker に監視したい対象またはイベントを動的に設定するために使う．

NumChecker の動作を，図 2.3 を用いて説明する．まず，ゲスト OS のユーザ空間にあるテストプログラムにより，システムコール A の呼び出しが発生したとする．制御をゲストのカーネルに移し，システムコール A のカーネル内処理ルーチンを行う前に，更に制御を KVM により[*2]ホストのカーネルに移す．次に，システムコール A が監視対象か否かを判断する．監視対象であるなら，HPCs の初期化を行いカウントを開始し，制御をゲストのカーネルに戻し，通常のカーネル内処理ルーチンを行う．最後に，システムコール A のカーネル内処理ルーチンの終了時に，iret 命令により，制御を再びホストのカーネルに移し，カウントを停止し，HPCs の値を読み込み，制御をゲストのユーザ空間のテストプログラムに戻す．

上で説明した通り，システムコールの実行前と実行後の CPU カウンタにある情報をゲストのカーネルから取り出すために，対象システムを一時的に止める必要がある．

VSyscall や NumChecker だけでなく，同様の方法を採用している研究は他にも複数ある．

例えば，VMscope [13] は，ゲスト側で動くプロセスとそのプロセスが呼び出したシステムコールの関連を調べるため，対象システムを一時的に止めてシステムコールの実行時にレジスタにある値を得る．

このように，ゲストからホストにある検知システムに情報を送る等の処理を行うため，対象システムを一時的に止める方法が一般的である．これに対して，本研究では対象システムを止めないため，外部から対象システムのメモリを直接訪問することにより，システムを実現する．

[*2] KVM の設定で，int 0x80 や iret 命令を敏感命令に設定することができる．

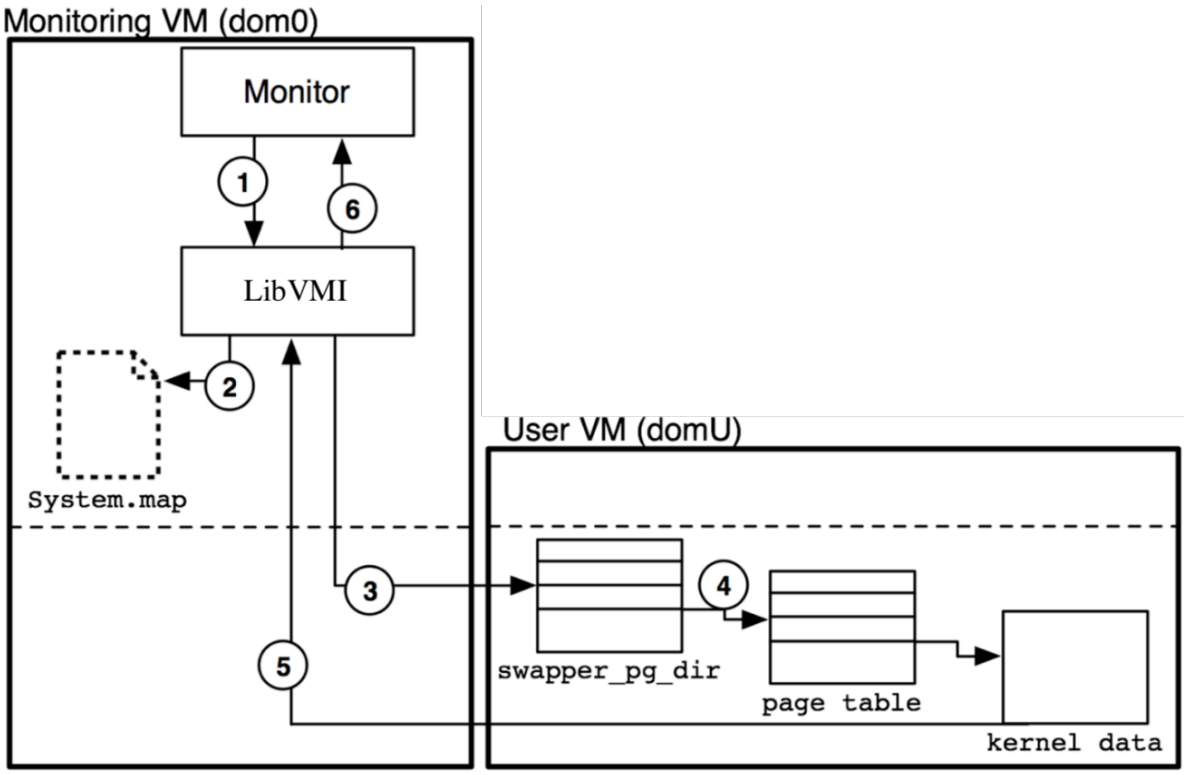


図 2.4　LibVMI の動作

2.2　仮想マシンを止めないシステム

LibVMI [15] は, Xen [16] 仮想マシンを対象として設計された Vitrual Machine Introspection (VMI) を行うためのツールである．これを用いれば，ゲスト OS (ドメイン U) を止めずにゲスト OS の情報を得ることができる．例えば，ゲストが発行したハードディスクに対する I/O 処理の監視や，ネットワークの監視などの機能を提供している．

LibVMI は，ホスト側 (ドメイン 0) の `xc_map_foreign_range` 関数を用いて，検知システムが欲しい情報を，ドメイン U から共有メモリ上に移す[*3]．LibVMI の動作を，図 2.4 に示す．LibVMI は，ドメイン U にある仮想アドレスをベースに，幾つかのテーブルを辿って，最終的に欲しい情報を見つけ，ドメイン 0 に移す．しかし，LibVMI は共有するメモリ量が限られているため，一度に移すことのできる情報の量が制限されている．このため，情報が大きいと，`xc_map_foreign_range` 関数の利用頻度が高くなり，コストがより大きくなるという欠点がある．ルートキットの存在を検知するためには，ゲスト OS から何度も情報を取得する必要があるため，コストが更に大きくなる．

[*3] この関数の中で，実質 `mmap` を呼んでいる．

3 Ftrace

本章では，Ftrace の用途及び基本的な使い方を述べる．そして，簡単な例を用いて，トレースの原理を説明する．最後に，実例を用いて，Ftrace の有用性について説明する．

3.1 基本的な仕組み

Ftrace は Linux カーネルに組み込まれているトレース機構である．その特徴は，動作中のカーネル内の関数呼び出しのイベントを記録し，ログとして出力することである．一般的に，カーネル内の関数の待ち時間やパフォーマンスの問題のデバッグや分析に使われている．

図 3.1 に示すように，Ftrace は以下の 4 つの部品で構成されている．

トレースポイント

トレースポイントは図 3.1 中の A に示した場所にある．その役割は，トレースしたい関数を実行する前に，トレース関数を先に呼び出すことである．トレースポイントは次の 2 つの処理で実現されている．1 つ目は，カーネルビルド時に，カーネル内のすべての関数定義の先頭にトレース関数を呼び出すためのアセンブリコードを埋め込み，更に，コードを埋めた場所を記録することである．ただし，この時に埋め込まれたコードは，カーネルイメージを作る時には，`nop` として書き換えられて，無効化されている．このようにすることで，デフォルトの状態ではトレース関数が呼び出されず，性能が損なわれない．2 つ目は，実際にトレースを行いたい時に

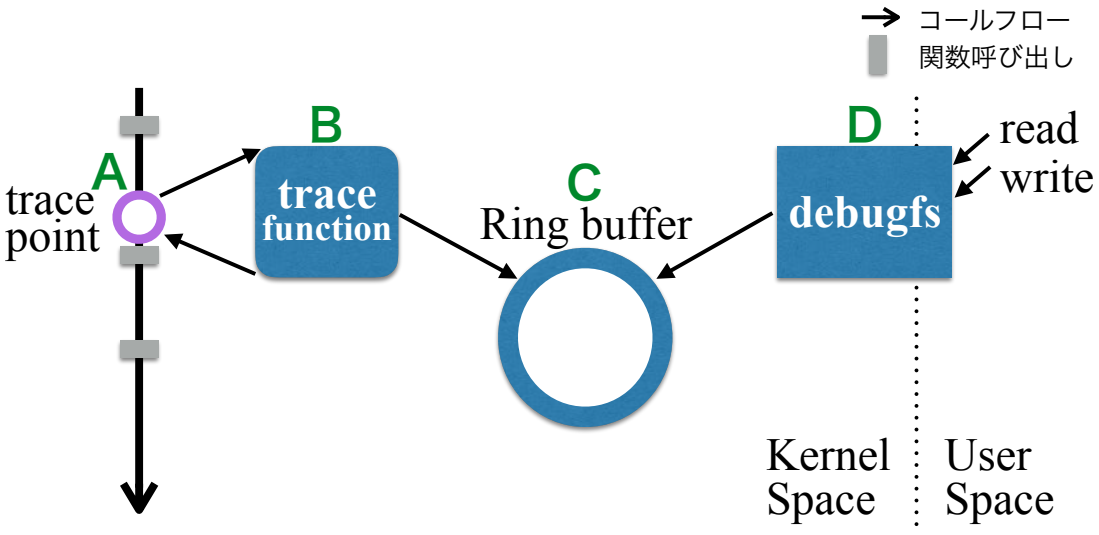


図 3.1　Ftrace の構造図

は，ユーザがトレースしたい関数にある nop を，トレース関数を呼び出すためのコードに書き換えることである[*1]．

トレース関数

トレース関数は図 3.1 の B にある．その役割は，ユーザが指定したトレースの機能を果たして，トレースの結果をリングバッファに送信することである．Ftrace は色々な機能をプラグイン (トレーサ) として，ユーザに提供している．Ftrace が提供しているトレーサの種類は，以下の命令で見ることがでる．

```
# cat available_tracers
blk mmiotrace function_graph wakeup_rt wakeup function nop
```

本研究では，コールフローを取得するため，function_graph トレーサを用いる．トレーサを指定する命令は，以下の命令で行う．

```
# echo function_graph > current_tracer
```

リングバッファ

リングバッファは図 3.1 の C にある．その役割は，トレース結果を保存することである．カーネルのリングバッファは，Ftrace だけでなく，様々なログ出力機構に使われている．そして，リングバッファへの書き込みの開始と終了の制御方法は，二種類がある．その一つはデバッグファイルシステムを用いる方法であり，以下の通りである．

```
# echo 1 > tracing_on
# echo 0 > tracing_on
```

また, カーネルモジュールでも利用可能とするため, もう一つの方法として, Ftrace は tracing_on と tracing_off API も用意している．

デバッグファイルシステム

図 3.1 の D に該当する．その役割は，ユーザはこの機構を用いて，Ftrace の設定を行うことができる．また，この機構を利用して，リングバッファに保存した情報を出力することである．トレースの結果は，trace ファイルに保存される．

[*1] Ftrace はブレイクポイントを用いて，コードの書き換えを行っている．

```
1  bar() {
2    foo();
3    // <- parent_ip
4    ...
5  }
6
7  foo() {
8    call ftrace(parent_ip, ip)
9    // <- ip
10   ...
11 }
12
13 ftrace (parent_ip, ip) {
14   prepare_function_retrun(...);
15     ...
16 }
17
18 retrun_to_handler() {
19   parent_ip = ftrace_return_to_handler(...);
20   ...
21 }
```

図 3.2　コールフローを取得する例

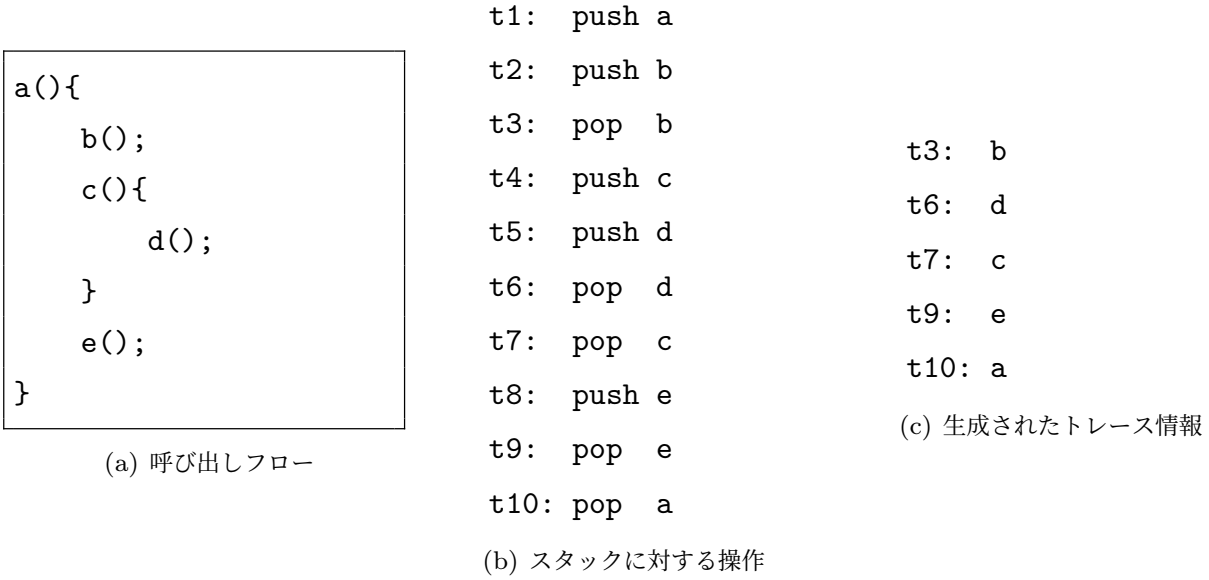
3.2　トレースの実現原理

本節では，図 3.2 にある例を用いて，Ftrace のトレースの原理を簡単に説明する．

この例では，bar の実行により，foo の呼び出しが発生している．Ftrace は foo の定義の先頭に call ftrace(parent_ip, ip) のようなアセンブリコードを埋め込んで，関数 ftrace の呼び出しによりトレースの機能を果たしている．

関数 ftrace には，2 つの目的がある．

1 つ目は，関数のトレース情報を出力させるため，Ftrace の機構に ftrace の引数を渡すことである．ftrace の引数は二つあり，それぞれは foo の実行終了時にカーネルのスタックに保存されている bar に戻るためのアドレス parent_ip と，ftrace の実行終了時に foo に戻るためのアドレス ip である．Ftrace の機構は，この二つのアドレスを記録することにより，関数のコールフローを出力している．Ftrace は，先に説明したアドレスだけでなく，foo の開始時刻と終了時刻を両方とも記録する．これらはいずれも，ftrace により実現される．



(a) 呼び出しフロー

(b) スタックに対する操作

(c) 生成されたトレース情報

図 3.3　トレーススタックの動作例

2 つ目の目的は，ftrace の実行により，カーネルのスタックに保存された parent_ip を Ftrace が用意した関数 return_to_handler に置き換えることである．この過程は prepare_ftrace_return により実現される．更に，prepare_ftrace_return は ftrace_push_return_trace を呼び出し，元の parent_ip や foo の開始時刻を Ftrace のトレーススタックに保存する．そして，foo の実行終了後，bar に戻ろうとする時，カーネルのスタックにある置き換えられた parent_ip に飛んで，return_to_handler を実行する．return_to_handler は，ftrace_pop_return_trace を呼んで，Ftrace のトレーススタックから，前に保存した情報に今の時刻を加え，リングバッファに保存し，元の parent_ip に飛んで，bar に戻る．最終的に，前節で述べだ通りに，トレースを生成し，ユーザに反映する．

また，カーネルビルド時に CONFIG_DYNAMIC_TRACER を有効にすると，動的トレースの機能が使えるようになる．この機能を用いると，動的にトレースの対象関数を変えることができる．これは，set_ftrace_filter により，トレース対象の関数を変更することができる．通常なら，トレースの対象はカーネル内全ての関数[*2]である．

Ftrace トレーススタック

Ftrace のトレーススタックは，トレース情報を生成するために，最も重要な部品である．図 3.3(a) にある例を用いて，Ftrace のトレーススタックについて説明する．この例では，Ftrace を有効にしたため，a が呼び出される時，本来 a からの帰り番地を保存すべきカーネルスタックに return_to_handler の番地を代わりに保存し，a からの帰り番地や呼び出し時刻 t1 をト

[*2] 例外が二つほどある，sched_clock と Ftrace 自身の関数はトレースの対象外である．

レーススタックに保存する．その後，処理が元に戻り，b が呼ばれる．先と同様に，カーネルスタックに Ftrace の return_to_handler のアドレスを入れ，b からの帰り番地と呼び出し時刻 t2 をトレーススタックに保存する．その後，b の処理に戻り，b が実行される．b の実行後，カーネルスタックに保存している return_to_handler に飛んで，return_to_handler の処理が実行される．return_to_handler では，現時刻 t3 (b の終了時刻) をトレーススタックに保存し，トレーススタックの情報 (b の帰り番地，呼び出しの深さ，実行開始時刻 t2，実行終了時刻 t3) を Ftrace のリングバッファに書き込む．その後，b の帰り番地に戻り，次に制御を c の実行に移す．

このように，本来の帰り番地を関数を呼び出す時に，Ftrace のトレーススタックにプッシュし，カーネルスタック上の帰り番地には return_to_handler の先頭アドレスを置く．関数の実行終了後には，return_to_handler に制御を移し，トレーススタックから情報をポップし，リングバッファに書き込み，本来の帰り番地に制御を戻す．トレーススタックに対する操作は，図 3.3(b) のようになる．最終的に，生成されたトレース情報が図 3.3(c) のように並べる．

3.3 Ftrace の有用性

Ftrace の有用性を，次の例を用いて説明する．この例では，open システムコールに与えられた引数を出力するようにシステムコールテーブルを書き換えている．

カーネルが正常な時の Ftrace の結果を図 3.4(a) に，システムコールテーブルが書き換えられている時の Ftrace の結果を図 3.4(b) に示す．この例の中で，Ftrace の結果を分かりやすくするため，通常のトレースに加えてシステムコール専用のトレース機能も併用した．この機能を利用すると，システムコールの引数も出力することができる．利用方法は，以下の通りである．

```
# echo 1 > events/syscalls/enable
```

二つのコールフローの結果を比較すると，open システムコールが怪しいのが分かる．図 3.4(b) では，本来の open システムコールによるカーネル内関数 sys_open の呼び出しが，アドレスが 0xffffffffc0b37000 である名前不明の関数の呼び出しに置き換えられている．これよりシステムコールテーブルが書き換えられている可能性が高いと判断することができる．

更に，前で述べた通り，Ftrace の利用は対象システムを止める必要がないため，本研究では，対象システムのトレース情報を取得する手段として，Ftrace を用いる．

```
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 0)   0.046 us    |    } /* down_read_trylock */
 0)               |  do_syscall_64() {
 0)               |    syscall_trace_enter() {
 0)               |    /* sys_open(filename:7f5ba2516ad0, flags:0, mode:0) */
 0)   0.191 us    |    }
 0)               |    SyS_open() {
 0)               |      do_sys_open() {
 0)               |        getname() {
 0)               |          getname_flags() {
 0)               |            kmem_cache_alloc() {
 0)   0.045 us    |              _cond_resched();
 0)   0.381 us    |            }
 ...
```

(a) 正常時

```
 0)               |  do_syscall_64() {
 0)               |    syscall_trace_enter() {
 0)               |    /* sys_open(filename:7ffe6072e100, flags:80000, mode:
                           7ffe6072e13)                                    */
 0)   0.521 us    |    }
 0)               |    0xffffffffc0b37000() {
 0)               |      printk() {
 0)               |        vprintk_func() {
                             ...
 0)    4.293 us   |              }
 0)   +10.759us   |            }
 0)   +11.113us   |          }
 0)   +11.540us   |        }
 0)   +11.909us   |      }
 0)               |      SyS_open() {
 0)               |        do_sys_open() {
 0)               |          getname() {
 0)               |            getname_flags() {
 0)               |              kmem_cache_alloc(){
 0)   0.050 us    |                _cond_resched();
 0)   0.562 us    |              }
 ...
```

(b) システムコールテーブルが書き換えられた時

図 3.4　Ftrace の結果から作成されたコールフロー

4 xftrace の設計

本章は xftrace の概要を示した後，まず Ftrace の利用について説明する．次に，Ftrace の利用により生じる問題点を述べる．最後に，問題の解決方法を含めた xftrace の設計を述べる．

4.1 Ftrace の利用

xftrace の設計における目標は，仮想マシンを止めずに，ゲスト側のカーネル関数呼び出しのコールフローをホスト側で出力することである．そのため，ゲスト側の Ftrace を用いて，目的を達成する．トレース結果は，ホスト側でゲスト OS のメモリを覗くことにより直接取得する．

本研究で用いるトレーサは，`function_graph` である．そして，デバッグファイルシステムを用いるトレースの開始と終了は，トレース時間を正確にコントロールするのが難しいため，本研究では，トレースの開始と終了のための API を利用する．更に，トレースの開始と終了は，ホスト側で制御できるように，システムを設計した．この部分は 4.4 節で詳細に説明する．

Ftrace は対象システムのトレース情報を取得するが，対象システムの内部にあるため，トレース情報がルートキットにより改竄される恐れがある．本研究では，マルウェアの攻撃によるゲスト OS のメモリの改竄に関して，次のような前提をおく．

- カーネルのコードセグメントの情報を改竄できない．
- カーネルのデータセグメントにある情報は改竄できる．

このため，本研究ではトレースポイントとトレース関数の利用は無効化されないものとする．一方，カーネルのデータセグエントにあるトレース情報は改竄される可能性があるものとする．たとえば，リングバッファにあるトレース情報を抹消したたち書き換えたりして，ルートキットが導入した `evil_open` のような関数呼び出しの痕跡を隠すような改竄が考えられる．また，出力側 (デバッグファイルシステム) に対して，攻撃者が用意したフィルタを仕掛けて，怪しい痕跡を隠すような攻撃手法も考えられる．

4.2 攻撃者による改竄の防止

ルートキットによるトレース情報の改竄を防止するため，関数呼び出しの履歴情報を直接得るための方法考え，幾つかの方法を考察した．

まず第一に，トレースの結果をリングバッファに送信せず，直接ホストに送信するように，システムを設計する方法が考えられる．この方法の問題点は，ゲストから外に情報を送信するために，対象システムを一時的に止める必要があるという点である．その理由は，2 節で述べた通りである．

第二は，xftrace が動作する間に，ゲスト OS のメモリ内で新たに割り当てられた領域にトレース結果を生成するようにする方法である．たとえば，Loadable Kernel Module (LKM) を用いて，トレース結果の送信先を置き換えることにより，この機能が実現できる．この方法の問題点は，LKM が既に攻撃者により汚染されている恐れがあるため，トレース結果の信用性が失われることである．

最終的に本研究では，バッファオーバーフロー攻撃に対する防御方法を参考として，トレース情報をリングバッファに書き込む際に，事前に用意した 64 ビットの鍵と排他的論理和 (XOR) をとってから書き込むように，xftrace の設計を定めた．この機能は Ftrace の機構を拡張することにより，容易に実現できる．鍵は，ホスト側で一定の時間間隔で新たに生成し，その鍵を直接ゲスト OS のメモリに書き込むことにより，ゲストに反映する．そして，ホスト側から，リングバッファにあるトレース情報を取得し，コールフローを生成する．しかし，リングバッファのデータ構造が複雑であるため，ホスト側でセマンティックギャップを埋めて，トレース結果を得るのは手間がかかる．このため，本研究ではリングバッファの代わりに，グローバル配列を用いる．

本システムに対し，攻撃者はトレース結果を偽装しようとしても，改竄すべき場所を調べ，更に異常が検知されないように当該の内容を書き換えるのが難しい．その理由が 3 つある．

1 つ目の理由は，ゲスト OS にある鍵は現在値しかなく，過去の鍵を持っていないためである．鍵はホストで短期間に更新されるため，攻撃者は過去のトレース情報を改竄しようとすると，過去の鍵の値を持つ必要があり，改竄が困難になる．

2 つ目の理由は，トレース情報をグローバル配列に書き込む順番は，関数の呼び出し順と異なるためである．3.2 節で説明した通り，Ftrace はトレーススタックを持ち，本研究ではトレーススタックからポップする時にトレース情報をグローバル配列に書き込むよう設計した．このため，マルウェアコードの間に正常な関数呼び出し関係のトレース情報が含まれ，攻撃者が仕掛けたマルウェアコードに関するトレース情報のみ削除するのが難しい．マルウェアコードのトレース情報とその間に挟まれる関数呼び出しを全部削除する改竄も考えられるが，その場合では，正常な関数のトレース情報が部分的になくなってしまう．

3 つ目の理由は，トレース情報に呼び出された関数の深さが含まれているからである．例えマルウェアコードのトレース情報を削除できたとしても，上下のトレース情報の中に関数の呼び出しの深さが含まれているため，最終的にコールフローを生成するとき，深さが不連続となり，怪しいことが明白になる．

しかし，この方法の利用により新たに生じる問題もある．ゲスト側における鍵の読み取りと，ホスト側で新たに生成した鍵の書き込みによる競合である．この競合により，XOR に用いる鍵の値として過渡的な値が読めて，結果としてトレース結果におかしな値が書き込まれる可能性がある．

この問題点への対処法は，以下 2 つのが考えられる．1 つ目は，ルートキットが存在することの判断を，一回だけのトレース結果からだけではなく，複数回トレースを分析することにより行

うことである．この競合が発生する頻度は低いと考えられるため，ルートキットの検知に及ぼす影響は少ない．2 つ目は，メモリ同期処理を行うことである．鍵をゲストのメモリに書き込む前に，そのメモリに対する書き込みができるか否かの判断を行う．これは，QEMU-KVM 内部にある機構により実現できる．しかし，この方法は，メモリ同期処理により発生するコストが大きい．本研究では，競合の可能性は高くないと判断し，1 つ目の方法を採用する．

4.3 コールフローの構築

ホスト側でコールフローを生成するため，以下の 3 つの情報が必要である．

a：ゲスト側のトレース情報

ホスト側から，ゲスト OS 内のグローバル配列にアクセスし，値を取得する．グローバル配列に格納した値は，以下の 4 つがある．

- 鍵と XOR をとった関数の仮想アドレス
- 関数の実行開始 (呼び出し) 時刻
- 関数の実行終了時刻
- 関数呼び出しの深さ

実行開始時刻と実行終了時刻は，対象システムにおける時刻である．これらは，対象システム内にあるグローバル変数 `jiffies_64` をベースとして計算した時刻で，単位はナノ秒である．関数呼び出し時の深さは，コールフローのインデクスを生成するために利用する．

b：ゲスト OS の System.map

ホスト側で，ゲスト OS のカーネルビルド時に生成したシンボルテーブル System.map を保持する．ここにはカーネルの関数やデータ構造とそれらの仮想アドレスが格納されている．図 4.1 に，System.map の一部を示す．

System.map の第一カラムは関数またはデータ構造の仮想アドレス，第二カラムはシンボルの型，第三カラムはシンボル名である．本研究で用いるシンボルの型は，以下の通りである．

- B または b: 未初期化データセクション
- D または d: 初期化済データセクション
- T または t: コードセクション
- R または r: 読み取り専用データセクション

我々は System.map を用いて，ゲストから取得した関数の仮想アドレスを関数名に変換す

```
...
ffffffff811c1900 T do_sys_open
ffffffff811c1b20 T sys_open
ffffffff811c1b40 T sys_openat
ffffffff811c1b60 T sys_creat
ffffffff811c1b80 T sys_close
...
ffffffff81d1c000 D jiffies
ffffffff81d1c000 D jiffies_64
...
ffffffff81f72a20 B global_trace
...
```

図 4.1　System.map の一部

る[*1]．マルウエアのコードのように対象システム起動後にロードされたカーネルモジュールに使われている関数は，System.map に含まれないため，生成されたコールフローには関数名ではなく，仮想アドレスがそのまま出力される[*2]．また，ホスト OS からの，ゲスト OS のメモリにあるグローバル配列や鍵のアドレスも System.map から取得する．詳しい説明は，第 5 章で行う．

c：鍵の履歴

ホスト側にある配列 Klog に，鍵と鍵を更新した時刻を履歴情報として保持する．ここでの時刻は，ゲスト OS のメモリにある `jiffies_64` を直接参照して計算した時刻である．この鍵の履歴は，トレース情報の関数の仮想アドレスと鍵の排他的論理和の値を正しいアドレスに戻すために利用する．

4.4　xftrace の構成

図 4.2，xftrace の全体像を示す．`global_key` は現在の鍵の値を保持するゲスト OS ないに変数であり，その初期値は 0 である．以下，xftrace を構成する部品について説明する．

A：Key_Controller

Key_Controller は鍵を生成するための部品であり，2 つの機能を果たしている．1 つ目は，一定時間間隔で新しい鍵を生成し，それを Klog に履歴として保存することである．2 つ目は，生

[*1] 主にシンボル型が T の行である．
[*2] Ftrace は，動的にロードされるカーネルモジュールにも対応するため，`/proc/kallsyms` を利用している．

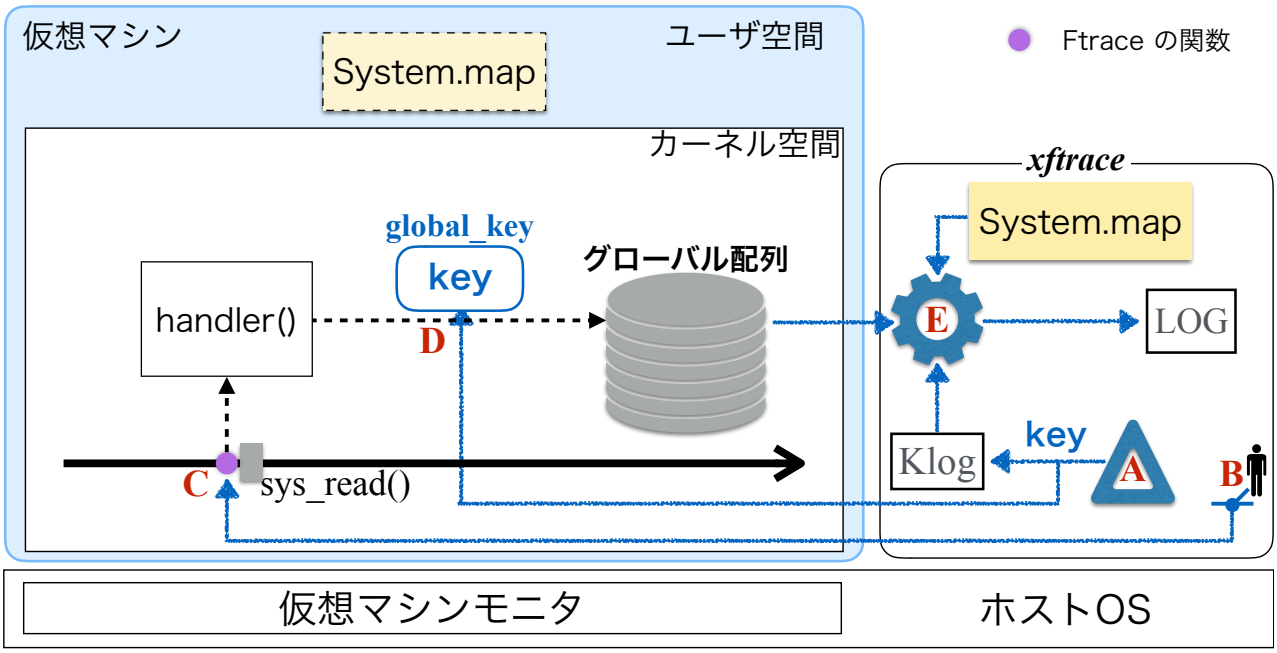


図 4.2　xftrace の概要

成した鍵をゲスト OS 内部の global_key に書き込むことである．鍵は 64 ビットの符号なし整数値である．

B と C：Trace_Switch

Trace_Switch は，トレースの有効化・無効化を行うための部品である．まず，ゲスト側で Ftrace のトレーサを `function_graph` に設定する必要がある．そして，ホスト側 (B) で，鍵を 0 から別の値に書き換えると，ゲスト側の Ftrace (C) が起動される．これは，ゲスト側で常に鍵の値を監視して，`tracing_on` を実行する部品を用いて実現した．ホスト側で，鍵を 0 に書き込むと，ゲスト側の Ftrace を停止する．これも上と同様に，`tracing_off` を実行する部品を用いて実現した．詳細の説明は 5 章で行う．

D：Push_Raw

Push_Raw は，ゲスト側の Ftrace のトレース情報と global_key の排他的論理和を取って，グローバル配列に書き込む．排他的論理和をとるタイミングは，3.2 節で説明した `ftrace_pop_return_trace` が実行する時である．同時に，関数の呼び出し時刻と関数の呼び出し時の深さをグローバル配列に書き込む．関数の実行終了時刻の情報を記録するタイミングは，`return_to_handler` を実行する時である．

E : Log_Maker

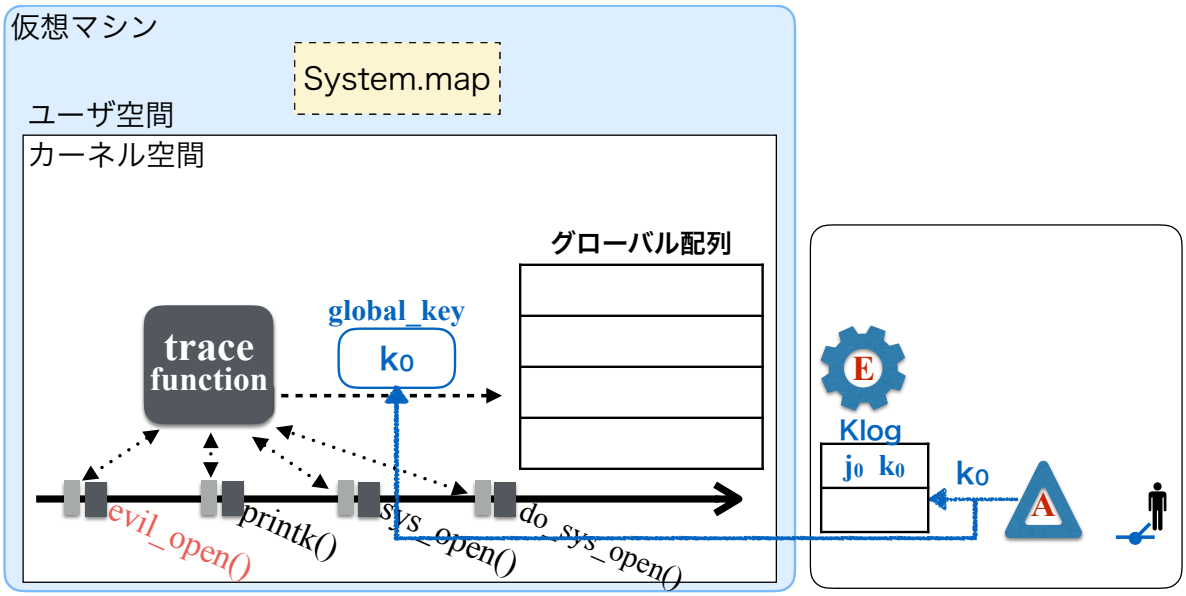
Log_Maker は，4.3 節で説明した部品 a，b，c を利用して，コールフローを生成する．

4.5 xftrace の動作例

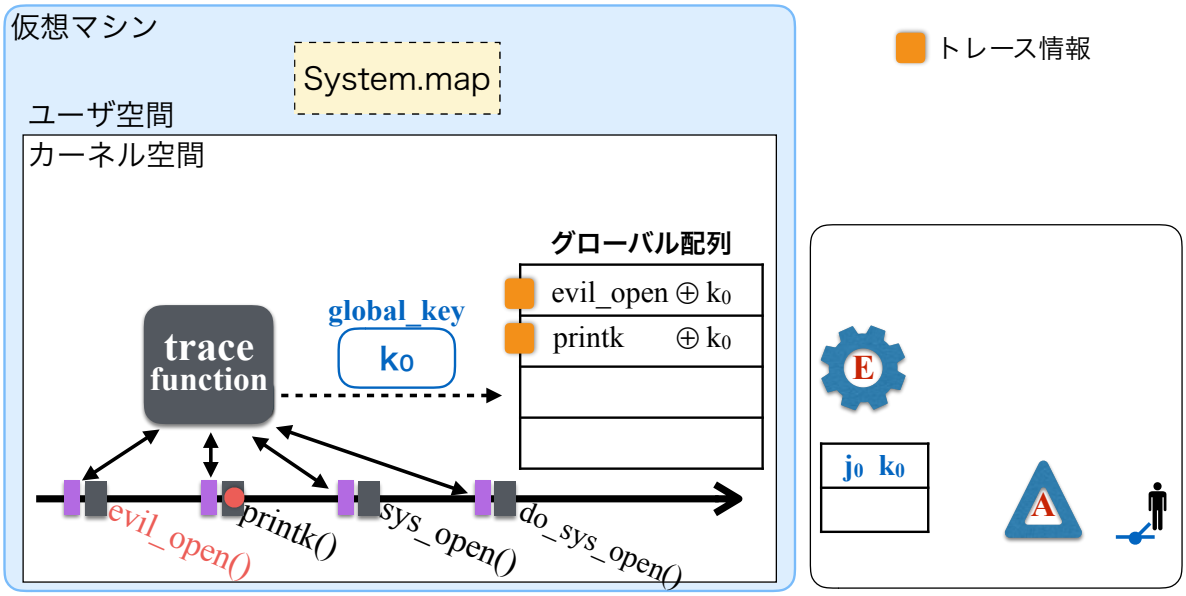
本節で，3.3 節で説明したシステムコールテーブルの open の処理関数のアドレスを書き換えた例を用いて，xftrace の動作を説明する．説明の都合上，下記簡略バージョンのトレース情報を用いる．

```
1  evil_open() {
2    printk();
3    sys_open();
4  }
5
6  sys_open() {
7    do_sys_open();
8  }
```

- ユーザはホスト側にある部品 A を用いて，鍵 (k_0) を生成する．
- 図 4.3(a) に示すように，k_0 を初期値が 0 である global_key に書き込む．同時に，k_0 とゲストの jiffes_64 の値 (j_0 とする) を鍵の値と共に，Klog に保存する．
- ホスト側の Trace_Switch を用いて，Ftrace を開始する．
- evil_open と printk を読んだ時，鍵の値が k_0 であるため，トレース関数が関数のアドレスと k_0 の XOR をとり，結果をゲスト側にあるグローバル配列に書き込む．その結果図 4.3(b) のようになる．
- 一定間隔がすぎたため，図 4.4(a) に示したように，新たな鍵 k_1 を生成し更新する．同時に，k_1 とゲストの jiffes_64 の値 (j_1 とする) を，Klog に保存する．
- 鍵の値が k_1 の間，sys_open と do_sys_open が実行されたとする，トレース関数が関数のアドレスと k_1 の XOR をとり，結果をグローバル配列に書き込む．
- xftrace が部品 E を用いて，ゲスト側のグローバル配列からトレース情報を取得し，Klog と System.map を利用して，コールフローを生成する．この過程を，図 4.4(b) に示す．



(a) 鍵の更新と登録



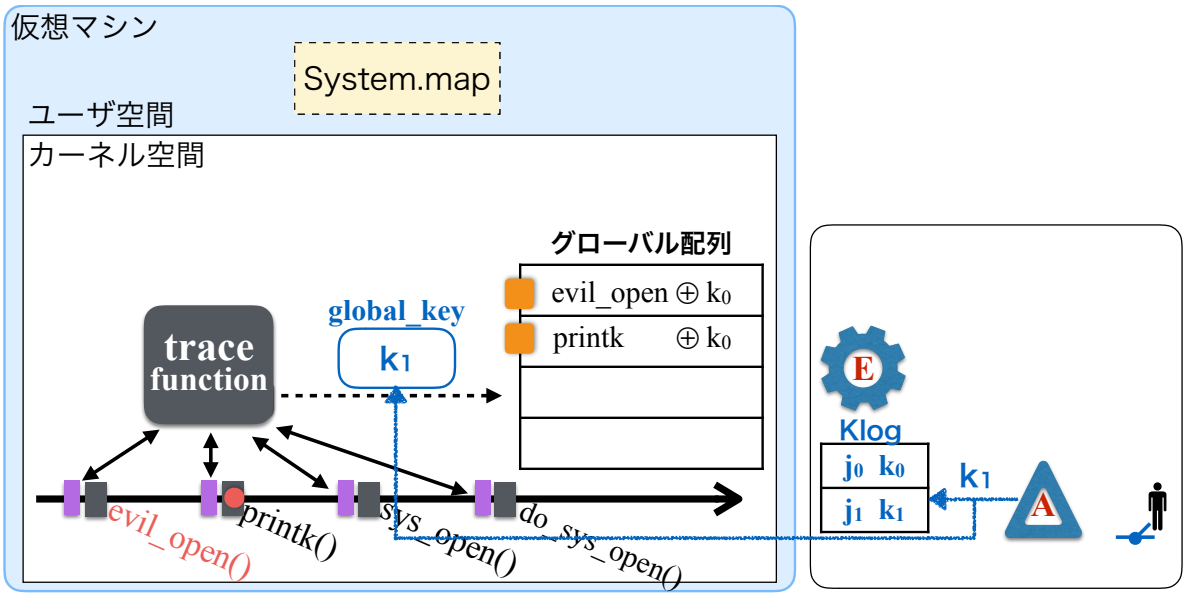
(b) 鍵が k_0 の間 xftrace の動作

図 4.3　xftrace の動作例 (1)

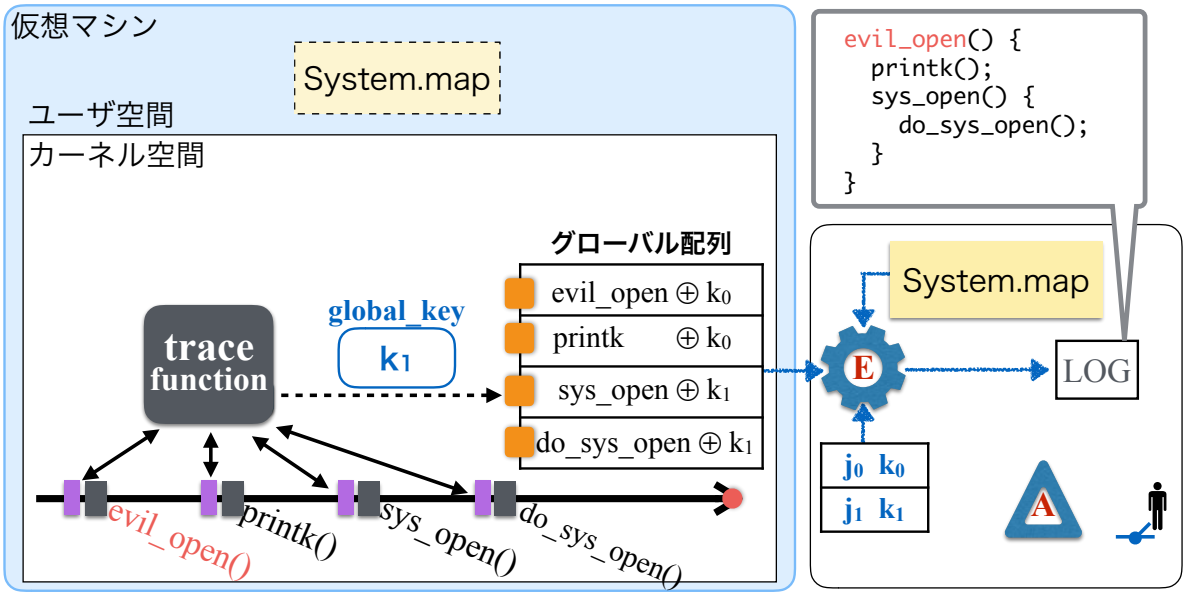
4.6　xftrace 利用方法

この節で，xftrace の利用方法について説明する．利用方法には，ユーザ主導方式とデーモン方式の 2 通りが考えられる．

ユーザ主導方式では，ユーザがゲスト側でテストプログラムを実行しながら，ホスト側で



(a) 2 回目の鍵の更新と登録



(b) コールフローの生成

図 4.4 xftrace の動作例 (2)

xftrace を動かす．テストプログラムでは，第 1 章で述べたよく狙われるシステムコールを呼ぶ．これはよく狙われるシステムコールの動作を調べることによる検出を目的とする．ユーザ主導方式を用いると，テストプログラムの実行により生成されるコールフローは一意であるはずなので，異常を検出しやすい．

ユーザ主導方式のメリットは，次の通りである．

- ユーザは，自分でトレース対象のシステムコールを選択することができるため，ユーザの自由度が高い．

- xftrace を常に有効にする必要がないため，ゲスト OS に対する負荷が少ない．

デーモン方式では，ホスト側のデーモンとして，xftrace を一定時間間隔で実行する方式である．通常時のゲスト OS の振る舞いを調べることによる検出を目的とする．

デーモン方式のメリットは，次の通りである．

- ユーザ主導方式のように，特定のプログラムの実行を想定していないので，より広い範囲のシステムコールの挙動を調べることができる．
- ユーザは，意識してプログラムを動かす必要がないため，ユーザの負担が少ない．

5 xftrace の実装

本章では，xftrace の実装方法について述べる．まず，仮想マシンのメモリに対して直接操作する方法を述べる．次に，xftrace の各部品を実現する方法について説明する．表 5.1 に示す環境を基とし，xftrace を実装した．

5.1 仮想マシンへの操作

4 章で述べたように，ゲスト OS 中のトレース情報の取得，一定時間間隔でのゲスト OS 中の鍵の更新などの目的のため，ホストからゲスト OS のメモリにアクセス手段が必要である．本節で，その実現手法について説明する．

5.1.1 ゲスト OS のメモリへのアクセス

本研究は KVMonitor [17] のメモリ操作方法を参考として，ホストからゲストのメモリを直接操作する機能を実現する．

本研究は，図 5.1 に示す通り，VM に割り当てる物理メモリをファイルとしてホスト上で作り，そのファイルを，VM と xftrace 用のプロセス空間に，`mmap` システムコールを用いてマップするようした．メモリファイルは QEMU-KVM が作成するバッキングストアを利用した．本来の QEMU-KVM では，他のプロセスがバッキングストアを利用できないように，`unlink` を用いて，バッキングストアファイルへのリンクが削除されている．VM メモリ割り当ての処理は，QEMU-KVM の `exec.c` の `file_ram_alloc` 関数で行われている．本研究は，バッキングストアを削除しないように，QEMU-KVM に変更を加えた．

更に，バッキングストアを生成する時に，書き込みと読み取り権限を与えた．QEMU-KVM 内では，`exec.c` の `file_ram_alloc` 関数内で，`mkstemp` 関数により読み取り専用のバッキン

表 5.1 実装環境

ホスト	メモリ	8GB
	OS	Ubuntu 14.04 x86_64
	QEMU-KVM	qemu-kvm 1.1.2
ゲスト	メモリ	1GB
	OS	Ubuntu 12.04 x86_64
	カーネル	Linux 3.13.0

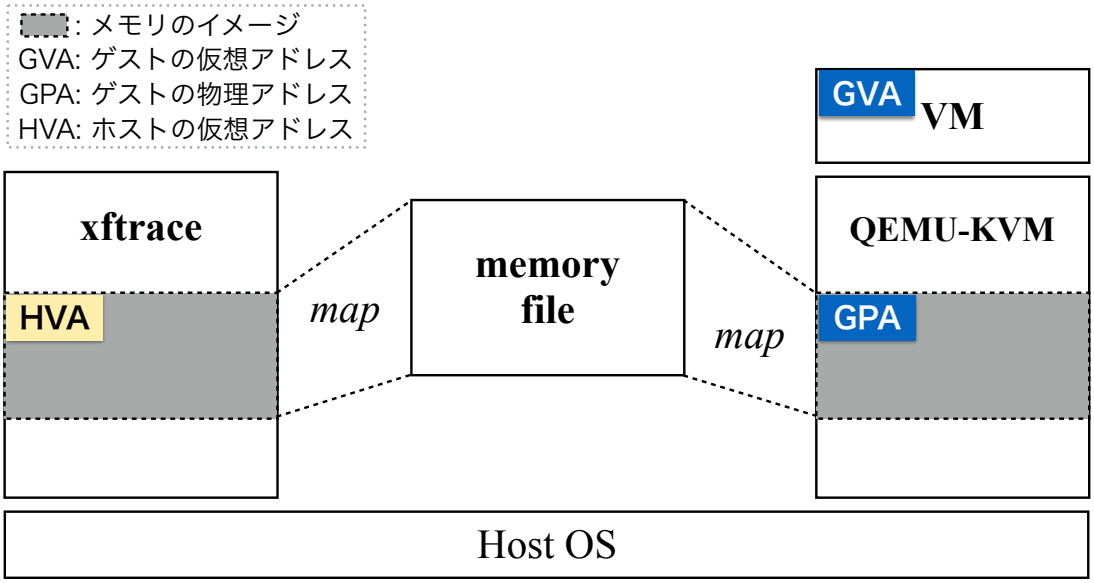


図 5.1　メモリ操作のイメージ図

グストアを生成してる．本研究では，生成されたバッキングストアに書き込み権限をあたえるため，mkstemp の代わりに，以下のコードを用いた．

```
fd = open(filename, O_CREAT | O_EXCL | O_RDWR, 0777);
```

更に，仮想マシンを立ち上げる時に，事前に指定した大きさのメモリを確保するように，以下のように QEMU-KVM のオプションで指定した．

```
$ qemu-system-x86_64 -hda /home/disk.img -m 1024  \
  -mem-path /hugepages -mem-prealloc
```

これは，ホスト上で連続した仮想アドレスを仮想マシンに割り当てるためである．-mem-path オプションでバッキングストアを格納するディレクトリを指定し，-mem-path オプションでメモリ領域の事前確保を行う．-m オプションで，仮想マシンメモリの大きさを 1024MB として指定した．

これにより，QEMU-KVM により生成される VM は従来の通りに動き，ホスト側から対象システムのメモリへの操作が VM を止めずにできるようになる．

5.1.2　アドレスの変換

ホスト側でゲストのメモリを操作するには，ゲストの仮想アドレス (GVA) をホストの仮想アドレス (HVA) に変換する必要がある．ゲストの物理アドレス (GPA) を経由して，アドレス変換では，まず GVA から GPA を計算し，次に GPA から HVA を計算する．その過程は図 5.2 で示す．

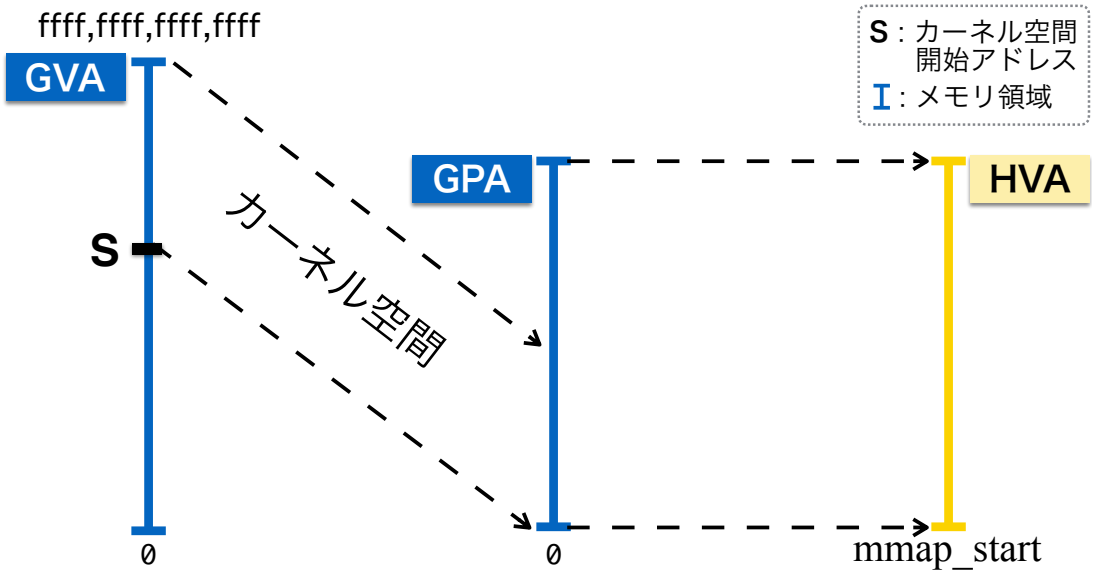


図 5.2　アドレス変換のイメージ図

GVA はカーネル空間とユーザ空間に分れているため，GVA から GPA の計算は，それぞれの空間に対して行う必要がある．カーネル空間にあるアドレスの変換は，仮想アドレスからカーネルのメモリ空間の開始仮想アドレスを引くだけで求まる．なぜなら，図 5.2 が示す通りに，カーネル空間は仮想アドレス上で連続に割り当てられていて，かつ物理メモリの 0 番地から始まるためである．本研究は 64 ビットの Linux 用いたため，カーネル領域の開始番地は `ffff800000000000` である．xftrace では，ユーザ空間に対して操作する必要がないため，その部分のアドレス変換のやり方は本論文から省略する．

GPA から HVA の計算も単純である．xftrace で，`mmap` システムコールを用いて，メモリファイルを VM と xftrace にマップしたため，GPA と HVA のアドレスは一対一対応になっている．このため，GPA から HVA へのアドレス変換は，マップの開始アドレス (図 5.2 にある `mmap_start`) をオフセットとして，計算する．

5.2　xftrace の実現

本節では，xftrace の実装をゲスト OS の拡張と xftrace 各部品の実装に分けて説明する．

5.2.1　ゲスト OS の拡張

グローバル配列と鍵の実装

図 5.3 にあるコードを用いて，グローバル配列のデータ構造を説明する．グローバル配列のメンバーは以下の 4 つがある．

- `func` は現在トレースされている関数のアドレスである．

```
1  struct graph_array {
2    unsigned long func;   /* Current function */
3    unsigned long long calltime;
4    unsigned long long rettime;
5    int depth;
6  };
7
8  #ifndef __XFTRACE
9  #define __XFTRACE
10 #define XFTRACE_MAX 20,000
11 extern unsigend long global_key;
12 #endif
13
14 extern struct graph_array xftrace_array[XFTRACE_MAX];
15 extern int xftrace_index;
```

図 5.3　グローバル配列のデータ構造

- calltime はトレースされている関数の呼び出された時刻である．
- rettime はトレースされている関数の実行終了の時刻である．
- depth はトレースされている関数の関数の呼び出し時の深さである．

グローバル配列の大きさは，20,000 に設定した．この数値は，通常の Ftrace のトレース結果から得られた．Ftrace により生成されたコールフローの行数は，二万行前後であるため，本研究で用いるグローバル配列の大きさを同じ程度になるように設定した．鍵を 64 ビットのグローバル変数として宣言し，0x0 として初期化した．また，グローバル配列にトレース情報を書き込むために，xftrace_index を利用している．これらの宣言は，linux/include/linux/ftrace.h で行い，定義は linux/kernel/trace/trace_functions_graph.c で行なった．

グローバル配列にトレース情報を書き込むため，functions_graph トレーサの拡張を行なった．linux/kernel/trace/trace_functions_graph.c の ftrace_pop_return_trace 関数に以下のコードを追加した．

```
1  if ((global_key != 0) && (xftrace_index < XFTRACE_MAX)) {
2     xftrace_array[xftrace_index]->func = global_key^trace->func;
3     xftrace_array[xftrace_index]->calltime = trace->calltime;
4     xftrace_array[xftrace_index]->depth = trace->depth;
5  }
```

この if 文で，鍵の値を確認し，鍵が 0 でない時に xftrace_array に鍵と XOR したトレース情報を書き込む．また，ftrace_return_to_handle に以下のコードを追加した．

```
1  if ((global_key != 0) && (xftrace_index < XFTRACE_MAX)) {
2     xftrace_array[xftrace_index]->rettime = trace.rettime;
3     xftrace_index+=1;
4     if (xftrace_index == XFTRACE_MAX) xftrace_index = 0;
5  }
```

この if 文で，2 つのことを行なっている．1 つ目は，xftrace_array に rettime を書き込んでいる．2 つ目は，xftrace_index を動かしている．コードに示した通り，xftrace_index が最大になった時，値が 0 に戻り，トレース情報が 0 番目の配列から書き込むようになっている．これは，ドレースの動作をグローバル配列の大きさで終了させないためである．

タイマ割り込みに対する変更

トレースの開始と終了の制御は，タイマ割り込みの変更により実現した．Linux カーネルでは，タイマ割り込みを用いて，時間の経過を記録している．本研究で，以下のコードを do_timer 関数に追加した．

```
1  if ((global_key != 0) && (global_trace.buffer_disable == 1)){
2      tracing_on();
3      xftrace_timeflag = 1;
4  } else if ((xftrace_timeflag == 1) && (global_key == 0) &&
5              (global_trace.buffer_disable == 0)) {
6      tracing_off();
7      xftrace_timeflag = 0;
8  }
```

この if 文で，タイマ割り込みが発生する時に，鍵の値を確認する．鍵の値が 0 から変化する時に tracing_on 関数を用いてトレースを開始し，鍵の値が 0 に戻る時に tracing_off 関数を用いてトレースを終了する．トレースが開始の状態なら，global_trace.buffer_disable の値が 0 であり，停止の状態なら，値が 1 である．

5.2.2 xftrace 各部品の実装

Key_Controller

Key_Controller を実現するため，ゲストのカーネルにグローバル変数 global_key を作る．

ホスト側で 64 ビットの鍵を生成する部分の実装を，図 5.4 に示す．struct klog には 2 つのメンバがあり，jiff は鍵を更新する時にゲスト OS の時間を保持し，key は生成された鍵を保持する．for 文で，KNUM 個の鍵をランダムに生成する．C ライブラリの rand 関数は，0 から 0x7FFF までの乱数しか生成できない．64 ビットの鍵を生成するため，12 ビットずつ生成するように実装した．そして，トレースを終了させるため，KNUM-1 番目の鍵を 0 とした．

```
1  struct Klog {
2     unsigned long long jiff;
3     unsigned long key;
4  }gkey[KNUM];
5
6  srand(time(NULL));
7  for(i = 0; i < KNUM-1; i++)
8     gkey[i].key =
9       ((unsigned long)rand())       & 0x0000000000000FFF | \
10      (((unsigned long)rand())<<12) & 0x0000000000FFF000 | \
11      (((unsigned long)rand())<<24) & 0x0000000FFF000000 | \
12      (((unsigned long)rand())<<36) & 0x0000FFF000000000 | \
13      (((unsigned long)rand())<<48) & 0x0FFF000000000000 | \
14      (((unsigned long)rand())<<60) & 0xF000000000000000;
15
16 gkey[KNUM-1].key = 0x0;
```

図 5.4　ホスト側鍵の生成

生成された鍵をゲストカーネル内のグローバル変数 global_key に直接書き込み，鍵と時刻をホストにある配列 Klog に記録する．この部分の実装を，図 5.5 に示す．2 行目から 4 行目は，ゲスト OS にある global_key と jiffes_64 の GVA を用いて，それらの HVA を計算する．mmap_start はメモリファイルを xftrace にマップする時の開始番地であり，KERNEL_TEXT_START はゲスト OS のカーネル領域の開始アドレスである．そして，一定時間毎に，鍵を更新するように for 文を用いた．8 行目のコードを用いて，ホスト側で生成した鍵をゲスト OS の global_key に書き込む．同時に，10 行目のコードで，ゲスト OS の時刻を gkey[i].jiff に記録する．最後に，12 行目の usleep を用いて，SUSPEND_TIME ナノ秒後に鍵を更新する．

Trace_Switch

5.2.1 節で説明した通り，Trace_Switch の機能はタイマ割り込みの拡張により実現した．ゲスト OS の Ftrace の tracer を function_graph に設定し，ホスト側で鍵を生成し書き込むと，トレースを開始する．また，鍵の値を 0 に戻すことにより，トレースを終了する．トレーサの指定は，ユーザが手動で行うか常にトレーサを function_graph にするかで行う．

Push_Raw

5.2.1 節で説明た通り，Push_Raw の機能はゲスト OS の Ftrace の function_graph トレーサを拡張することにより実現した．通常の Ftrace では，ftrace_return_to_handler 関数の

```
1  // caculate global_key address
2  p_gkey = mmap_start + ADDR_GKEY - KERNEL_TEXT_START;
3  // caculate jiff address
4  p_jiff = mmap_start + ADDR_JIFFES - KERNEL_TEXT_START;
5
6  for (i = 0; i < KNUM; i++) {
7      // put gkey into global_key
8      memcpy(p_gkey, &gkey[i].key, sizeof(gkey[0].key));
9      // get guest side time (jiff)
10     memcpy(&gkey[i].jiff, p_jiff, sizeof(gkey[0].jiff));
11     // suspend time
12     usleep(SUSPEND_TIME);
13 }
```

図 5.5　ゲスト OS にある鍵の更新

実行により，リングバッファにトレース結果を記録する．xftrace では，この関数が呼び出される時に，呼び出された関数のアドレスと global_key の排他的論理和を取ってから，グローバルの配列に書き込むようにする．

Log_Maker

Log_Maker は，ゲスト側にあるトレース情報，ゲスト OS の System.map，鍵の履歴の 3 つの情報を用いて，ホスト側でコールフローを生成する．

まず，ホスト側でゲスト OS にあるグローバル配列から，直接トレース結果を取得する実装について，下のコードを用いて説明する．

```
1  unsigned long guest_array[ARRAY_MAX*MEB];
2  p_array = mmap_start + ADDR_ARRAY - KERNEL_TEXT_START;
3  memcpy(guest_array, p_array,
4         sizeof(guest_array[0])*ARRAY_MAX*MEB);
```

ホスト側でゲスト OS にあるグローバル配列と同じ大きさを持つ配列 guest_array を宣言する．ゲスト側 global_array の HVA を計算し，global_array にあるトレース情報を guest_array にコピーする．

次に，トレース情報と鍵の XOR をとり，元の関数アドレスに戻す処理について説明する．履歴にある鍵を得られたトレース情報と XOR を行い，得られた値が System.map にあるシンボルの仮想アドレスのひとつと一致すれば，その仮想アドレスを保存する．この時，鍵は gkey[0].key，gkey[1].key，... の順でしか使われないことを顧慮した処理を行う．

最後に，System.map を用いて，トレース情報のアドレスを関数名に変換し，コールフローを

生成する．出力をコールフローの形にするため，トレース情報を `calltime` 順でソートし，各関数のインテントをトレース情報の `depth` 分とするように実装した．また，コールフローを生成する時に，現在の関数と次の関数の深さが 2 以上増えることはありえないため，深さが 2 以上増えることが見つかったらエラーメッセージをコールフローの間に出力するように，実装した．

6 xftrace の評価と議論

本章で，xftrace の検知能力と xftrace の使用により発生するオーバーヘッドについて，評価を行う．最後に攻撃の可能性及び今後の発展について議論を行う．

評価のための実験環境は表 6.1 に示す．

6.1 検知能力の評価

検知能力の評価は，3.3 節で利用した例を利用する．open システムコールの通常時のコールフローと，LKM を用いて evil_open でシステムコールの処理ルーチンを置き換えた時のコールフローを比較する．evil_open は，open システムコールの引数を printk を用いて表示した後，sys_open を呼び，通常のオープン操作を行う関数である．通常通りに動作する open のコールフローの結果を図 6.1(a) に，システムコールテーブルが書き換えられている時のコールフローの結果を図 6.1(b) に示す．

二つのコールフローを比較すると，open システムコールが怪しいのが分かる．図 6.1(b) では，本来の open システムコールによるカーネル内関数 sys_open の呼び出しが，アドレスが ffffffffa00cb000 である関数の呼び出しに置き換えられていることがわかる．関数 evil_open が System.map に存在しないため，関数のアドレスが出力されている．

また，コールフローの内部で，関数と次の関数の深さが 2 以上増えるエラー (5.2.2 節参照) が，図 6.2 のように出現した．通常の Ftrace のトレース結果においても，深さコールフローに同様のずれが生じるこのから，この問題は Ftrace の問題と判断できる．しかし，この現象は発生す

表 6.1 実験評価に使用したシステムの構成

ホスト	CPU	Intel Core i7-3770 3.40GH
	メモリ	8GB
	OS	Ubuntu 14.04 x86_64
	カーネル	Linux 3.2.0
	QEMU-KVM	qemu-kvm 1.1.2
ゲスト	メモリ	1GB
	OS	Ubuntu 12.04 x86_64
	カーネル	Linux 3.13.0

```
SyS_open() {
  do_sys_open() {
    getname() {
      getname_flags() {
        kmem_cache_alloc() {
          _cond_resched();
        }
      }
    }
    get_unused_fd_flags() {
      __alloc_fd() {
        _raw_spin_lock();
        expand_files();
        _raw_spin_unlock();
      }
    }
    do_filp_open() {
      path_openat() {
        get_empty_filp() {
          kmem_cache_alloc() {
            _cond_resched();
          }
...
```

(a) 正常時

```
ffffffffa00cb000() {
  printk() {
    vprintk_emit() {
      _raw_spin_lock();
      log_store();
      console_trylock() {
        down_trylock() {
        ...
      }
    }
  }
  SyS_open() {
      do_sys_open() {
      getname() {
        getname_flags() {
          kmem_cache_alloc() {
            _cond_resched();
          }
        }
      }
      get_unused_fd_flags() {
        __alloc_fd() {
...
```

(b) システムコールを置き換えた時

図 6.1　xftrace により生成したコールフロー

る頻度が低い[*1]ため，ルートキットの分析に支障がでないと判断する．

この評価により，xftrace のルートキットの検知能力が確認できた．ただし，攻撃者の攻撃コードに，通常のカーネルの関数の呼び出しが発生しない場合が考えられる．xftrace では，このような攻撃の検知ができない．

6.2　オーバーヘッドの評価

以下の場合について，テストプログラムの実行時間を比較することにより，xftrace の利用により発生するオーバーヘッドの評価を行った．

- Ftrace なしの場合
- トレーサのみ指定した場合 (トレーサを function_graph に指定し，トレース情報を書き

[*1] 20,000 個のトレース情報中に 5 回ぐらいしか出ない

```
...
          update_stats_wait_end() {
/*func:update_stats_wait_end depth[32]:5 depth[33]:16 d: 11*/
                                  finish_task_switch();
...
    __sb_start_write() {
/*func:__sb_start_write depth[9]:2 depth[10]:5 d: 3*/
          mutex_lock() {
...
```

図 6.2　コールフローにあるエラーメッセージ

表 6.2　評価プログラムの実行時間 (単位:s)

なし	トレーサのみ	Ftrace	xftrace
0.9272	5.1046	36.2422	36.5566

込まない状態)

- 通常の Ftrace の場合 (トレーサを function_graph に指定し，トレース情報を書き込む状態)
- xftrace の場合

実験で用いたテストプログラムは，open，close システムコールを 1,000,000 回ずつ直接呼ぶものである．結果を，表 6.2 に示す．この実験結果より，Ftrace と xftrace の利用により発生するオーバーヘッドの差が小さいことがわかる，xftrace の利用時，ゲスト OS に負荷を掛けるが，xftrace の 1 回の利用時間が短ければゲスト OS に与える影響は小さい．

しかし，xftrace の現状として，トレーサの指定はゲスト OS 内部でした行えない．このため，xftrace をデーモン方式で動作させると，ゲスト OS で事前にトレーサを function_graph に設定する必要がある．表 6.2 に示した通り，トレーサのみ指定した場合でもオーバーゲッドがかかるので，xftrace によるトレース情報の取得を行われなくても同程度のオーバーヘッドはかかってしまう．

6.3　議論

本節で，攻撃の可能性及び今後の発展について議論をする．

```
evil_open(){
  printk();
  getpid(){
    unlink();
  }
  sys_open();
}
```

(a) 呼び出しフロー

```
1: printk
2: unlink
1: getpid
1: sys_open
0: evil_open
```

(b) 生成されたトレース情報

図 6.3　攻撃例その 1

6.3.1　攻撃の可能性

マルウェアにより，トレース情報等が改竄される可能性について議論する．攻撃者が痕跡を隠し，トレース情報を改竄するには，以下 2 つの条件を満たす必要がある．

- 全ての鍵を取得すること．
- 正しいコールフローの形を偽造すること．

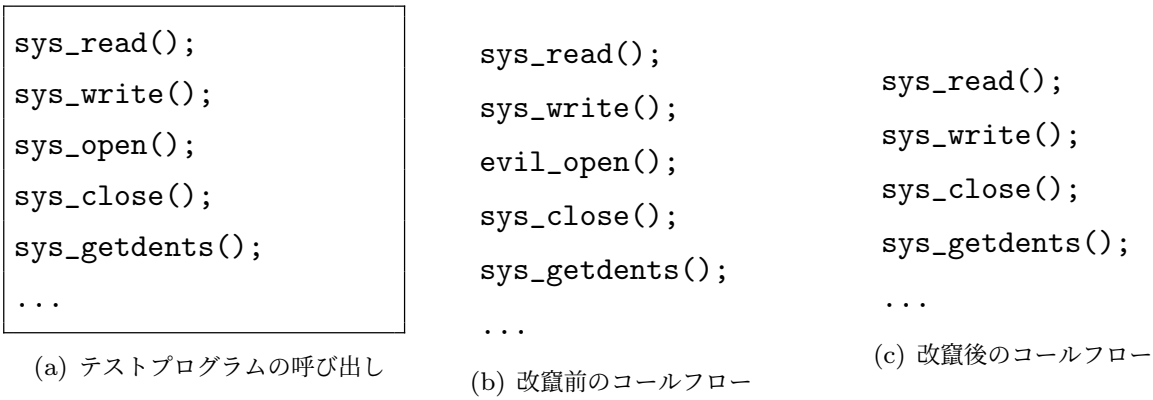
鍵はゲスト OS のメモリ空間にあるため，攻撃者が工夫して入手することはありえるが，トレース情報を改竄してコールフローを正常の形に偽造することが難しい．そこで，攻撃者が改竄する時，カーネルデータセグメントにあるトレース情報を保持するグローバル配列と Ftrace のトレーススタックが，攻撃の対象として狙われやすい．

グローバル配列を狙う攻撃

グローバル配列を対象とした，改竄を行う攻撃が考えられる．この時，evil_open のような関数の実行中に，自分自身のトレース情報を削除するような改竄と，グローバル配列にすでに書き込まれているトレース情報を改竄する 2 つの場合が考えられる．

自分自身のトレース情報を削除するような改竄は，トレース情報をグローバル配列に書き込む順番は関数の呼び出し順と異なるため，痕跡を残さずに改竄することは難しい．図 6.3 にある例を用いて，これを説明する．攻撃者は evil_open で open システムコールの処理ルーチンを置き換えたとする．この時，グローバル配列にあるトレース情報が図 6.3 が示す順に書き込まれる．グローバル配列中の evil_open に関連するトレース情報を消すなら，evil_open 関数とその中で呼ばれた printk，getpid，unlink のトレース情報全部消し，更に sys_open を残す必要がある．トレース情報が呼び出し順ではないため，このような改竄を行うのは易しくはない．

グローバル配列にすでに書き込まれたトレース情報を改竄する攻撃を困難にするために，本研究ではゲスト OS に一定時間毎に変化する鍵を用意して，トレース情報と排他的論理和をとるよ



(a) テストプログラムの呼び出し

(b) 改竄前のコールフロー

(c) 改竄後のコールフロー

図 6.4　攻撃例その 2

うにした．ゲスト OS は現在の鍵しか保持していないため，使われた鍵の値が現在の鍵の値とは異なる過去トレース情報を矛盾なく改竄するのは難しい．また，過去のトレース情報を現在のトレース情報で上書きする場合では，「ユーザ主導方式」を用いれば，テストプログラムのコールフローに異変が生じるため，攻撃されたことがわかる．たとえば，図 6.4 の例で，攻撃者は `evil_open` で `open` システムコールを置き換えたとする．ユーザ主導方式において実行するプログラムで，図 6.4(a) に示す順でシステムコールを呼び出せば，図 6.4(b) のようなコールフローが生成される．しかし，過去のトレース情報を現在のトレース情報で上書きする攻撃を受けると，図 6.4(c) が示すように `sys_write` と `sys_close` が連続に現れ，`sys_open` が消えてしまうようなコールフローが生成され，攻撃されたことがわかる．

Ftrace のトレーススタックを狙う攻撃

Ftrace のトレーススタックを改竄の対象とする場合もある．この場合は，現在のトレーススタックにあるマルウェアの関数の帰り番地を消すような攻撃が考えられる．しかし，トレーススタックの帰り番地が消えてしまうと，3.2 節で説明したように，関数の実行が終了する時に，制御を元に戻せなくなってしまう．

xftrace を利用することにより，このような改竄を完全に防ぐことはできないものの，改竄の難易度を上げている．その結果，ルートキット検知の可能性を上げることができる．

6.3.2　今後の発展

Kernel Address Space Layout Randomization

カーネルのセキュリティを高める技術 Kernel Address Space Layout Randomization (KASLR) [23] を本研究と組み合わせると，攻撃者からの改竄が更に難しくなる．KASLR は，Linux 3.14 からサポートされた技術であり，通常の System.map にあるアドレスをそのまま使うことなく，カーネルがブートの度に異なるアドレスにロードするようになる．xftrace と共に

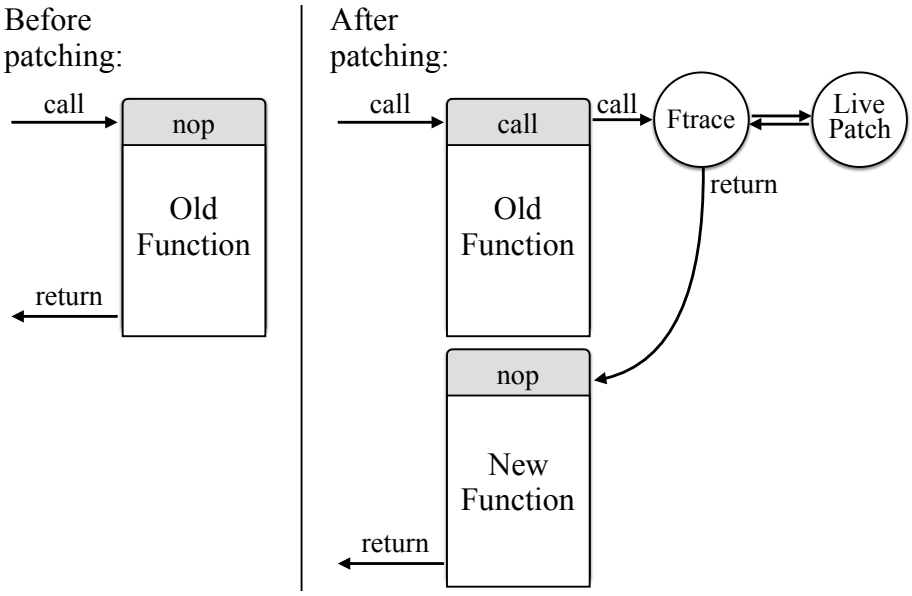


図 6.5　Kernel Live Patching のイメージ図

利用すると，改竄対象のアドレスを調べる手間がかかり，攻撃を仕掛けるのが難しくなる．

現状では xftrace が直接 System.map にある関数とアドレスの対応デーブルを用いて，コールフローを出力する．xftrace の KASLR への対応方法としては，ブート時に各関数のオフセットを計算し，xftrace 用のシンボルテーブルを生成し直すことである．

KASLR は Linux 4.12 からデフォルトで，使うようになっているが，Debian 系の Ubuntu や Redhat 系の CentOS などでは，まだ使用していない．

Kernel Live Patching

Kernel Live Patching [24] は，再起動せずにカーネルに新たな機能をライブパッチを行うための技術である．Kpatch [25] と Kgraft [26] がこれを実現したプロジェクトである．この技術は，一般的にサーバとして動作するシステムを止めずに，緊急のセキュリティパッチを適応するために使われている．Kernel Live Patching は，図 6.5 に示すように，Ftrace のメカニズムをベースとして実装された．Old Function が呼ばれた時に，Ftrace の機構により，制御が New Function に移るようにする．

この技術が攻撃者に用いられると，任意の場所にマルウェアのコードを設置されることができるようになる．しかし，Kernel Live Patching を利用しても，New Function から呼ばれるカーネル関数にトレースポイントが入っていると，New Function の動作がコールフローに反映されるようになる．これにより，コールフローの異常が検知できる．

7 関連研究

KVMonitor [17] は，KVM 仮想マシンを対象として，ゲスト上で動く検知システムをホスト上で動かすためのツールである．検知システムをホスト上で動かすため，ホスト側で検知システムが必要な情報を提供する必要がある．例えば，ゲストのメモリから再現したプロセスリストの情報，ハードディスクにあるファイルの情報，ネットワークに関わる通信パケットの情報等を KVMonitor で提供できるように設計されている．本研究は，ゲストにあるカーネル関数呼び出し履歴を収集するため，ゲストのメモリを直接覗き見る機能が必要である．本研究は，この点に関して，KVMonitor を参考した．

CFI [18] は，コールフローのインテグリティをチェックする研究である．CFI を用いると，ソフトウェアが乗っ取られているか否かの判断ができる．CFI では，監視対象の粒度の設定について，トレードオフが生じる．細かい粒度に設定すると，パフォーマンスが損なわれる．逆に大きい粒度に設定すると，検知能力が限られて，攻撃するのにに十分の隙間が生じる．xftrace は，カーネル関数を対象として，検知を行う時のみ Ftrace を動作させるため，常に起動状態である CFI よりコストが小さい．

SecVisor [19] と OSck [20] は，VMM を使ってカーネルのコールフローのインテグリティを保証する手法である．SecVisor は，セキュア OS を用いて，ユーザが意図しないカーネルへの変更を防ぐことができる．しかし，この方法では，ユーザの自由度がかなり制限されてしまうという問題が生じる．OSck は，カーネルのインテグリティを保証するため，静的にカーネルのコードを分析する手法と，動的にポインタが指すオブジェクトの型を分析する手法を用いている．しかし，カーネルのコードを分析する時に，仮想マシンの情報を得るため，対象システムを止める必要がある．これに対して，本研究では，それらの問題が存在しない．

DevOps [21] は，クラウドの上で稼働するシステムの性能を劣化させることなく，VMI を行うためのシステムである．対象システムに及ぼす影響を減らすため，DevOps はライブマイグレーションを用いて，対象システムのクロンを行い，クロンされたシステムに対する VMI を行う．しかし，ライブマイグレーションを行うため，僅かの間だけ対象システムを停止させている．本研究は，異なるアプローチを用いて，対象システムを停止することなく検知を行う．

Virtio-trace [22] は，ホスト側で，仮想マシンに関連するデバッグ情報を出力する．Virtio-trace はゲスト OS とホスト OS で取得したトレースデータをマージし，時間順で並べて表示することで，ゲスト OS 上で発生した問題に対して，ゲストまたはホストの原因の判断を容易にするという特徴がある．しかし，ゲスト OS から取得したトレースデータはルートキットに感染した場合には信用できないため，本研究の目的に Virtio-trace を利用することはできない．

8 おわりに

本論文では，VM を止めずに，VM の外からカーネル制御フロー改変ルートキットを検知するためのシステム xftrace を提案し実装した．本研究で，まずカーネル関数のコールフローをとることが，カーネル制御フロー改変ルートキットを検知するのに有効であることを説明した．xftrace は，QEMU-KVM のメモリファイルと拡張した Ftrace を用いて，VM を止めないでトレースを取ることを目指している．更に，攻撃者によるトレース情報の改竄を防ぐため，VM の上で動く Ftrace の結果 (カーネル内関数のアドレス) と 64 ビットの鍵の値との排他的論理和をとり，ホスト側でコールフローを生成するようシステムを設計した．xftrace は，従来の手法と異なり，ホスト側でゲストのプロセスリストのような情報を再構築する必要がなく，VM を止める必要がない．更に，一回の利用に当てる時間を短く設定すれば，xftrace の使用により生じるオーバーヘッドを小さく抑えることができる．

現状では得られたコールフローからの異常の検知はユーザが目視で行う．単純なコールフローなら，ユーザが容易に異常の検知をできるが，複雑な場合はユーザに大きな負担がかかる．このため，異常の検知を自動化することが，今後の課題である．

謝辞

本研究を行うにあたり，終始変わらぬ御指導を賜りました岩崎英哉教授，中野圭介准教授に深く感謝致します．また，日常の議論を通じて多くの知識や示唆を頂いた研究室の皆様に感謝しますまた，KVMonitor を提供して下さった九州工業大学の光来健一准教授に感謝致します．また，本論文をまとめるにあたり有益なコメントを下さった東京農工大学の山田浩史准教授，東洋大学の浅野智之助教に深く感謝いたします．

参考文献

[1] chkrootkit, http://www.chkrootkit.org/

[2] RootKit Hunter, http://rkhunter.sourceforge.net/

[3] 2001, D. D. Zovi, "Kernel Rootkits:SANS Institute InfoSec Reading Room", https://www.sans.org/reading-room/whitepapers/threats/kernel-rootkits-449

[4] J. N. L. Petroni and M. Hicks, "Automated detection of presistent kernel control-flow attacks", in Proc. 14th ACM Conf. Comput. Commun. Secureity, Alexandria, VA, USA, Oct. 2007, pp. 103-115.

[5] May 30. 2012, The Adore-ng Rootkit, https://stealth.7350.org

[6] July. 2017, Symantec, Internet Security Threat Report, https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/istr-living-off-the-land-and-fileless-attack-techniques-en.pdf

[7] 2008, Ftrace - Function Tracer, https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[8] QEMU-KVM, https://wiki.qemu.org/Features/KVM

[9] A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, "kvm: the Linux virtualmachine monitor." In: Proceedings of the Linux symposium. vol. 1, 2007, pp. 225-230

[10] Sd and Devik, "Linux on-the-fly kernel patching without LKM.", Phrack Mag., vol. 11, Jan. 2014.

[11] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, R. Sion, "Introspections on the Semantic Gap", IEEE Security & Privacy Volume: 13, Issue: 2, Mar.-Apr. 2015, pp. 48-55

[12] B. Li, J. Li, T. Wo, C. Hu, L. Zhong, "A VMM-based System Call Interposition Framework for Program Monitoring", Parallel and Distributed Systems , 2010 IEEE 16th International Conference, 8-10 Dec. 2010

[13] X. Jiang, X. Wang, "Öut-of-the-boẍMonitoring of VM-based High-Interaction Honeypots", RAID'07 Proceedings of the 10th international conference on Recent advances in intrusion detection, Gold Goast Australia, September 05- 7, 2007, pp. 198-218

[14] X. Wang, R. Karri, "Reusing Hardware Performance Counters to Detect and Identify Kernel Control-Flow Modifying Rootkits", 2016 IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp. 485-498

[15] B. D. Payne, M. D. P. de A. Carbone, W. Lee, "Secure and Flexible Monitoring of Virtual Machines", Computer Security Applications Conference, 10-14 Dec. 2007.

[16] P. Barham et al., "Xen and the art of virtualization", in Proc. 19th ACMSOSP, New York, NY, USA, 2003, pp. 164âĂŞ177.

[17] K. Kourai, K. Nakamura, "Efficient VM Introspection in KVM and Performance Comparion with Xen", 2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing, pp. 192-202

[18] M. Abadi, M. Budiu, U. Erlingsson, J. Ligatti, "Control-Flow Integrity Principles, Implementations, and Applications", ACM Transactions on Information and System Security, Volume 13 Issue 1, October 2009 Article No. 4

[19] A. Seshadri, M. Luk, N. Qu, A. Perrig, "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes", SOSP '07 Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, pp. 335-350

[20] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, E. Witchel, "Ensuring Operating System Kernel Integrity with OSck", ACM SIGARCH Computer Architecture News - ASPLOS '11, Volume 39 Issue 1, March 2011, pp. 279-290

[21] S. Suneja, R. Koller, C. Isci, E. Lara, A. Hashemi, A. Bhattacharyya, C. Amza, "Safe Inspection of Live Virtual Machines", VEE '17, Volume 52 Issue 7, July 2017, pp. 97-111

[22] virtio-trace: Support virtio-trace, https://lwn.net/Articles/508063

[23] Kernel address space layout randomization, https://lwn.net/Articles/569635/

[24] Kernel Live Patching, https://lwn.net/Articles/619390/

[25] kpatch ,https://git.kernel.org/pub/scm/linux/kernel/git/jirislaby/kgraft.git/

[26] kgraft , https://git.kernel.org/pub/scm/linux/kernel/git/jirislaby/kgraft.git/